The Dangers and Complexities of SQLite Benchmarking

Dhathri Purohith Jayashree Mohan Vijay Chidambaram

Department of Computer Science, University of Texas at Austin
email :{ dhathri, jaya, vijay} @cs.utexas.edu

Abstract

Benchmarking systems in a repeatable fashion is complex and error-prone. The systems community has repeatedly discussed the complexities of benchmarking and how to properly report benchmarking results. Using the example of SQLite, we examine the current state of benchmarking in industry and academia. We show that changing just *one* parameter in SQLite can change the performance by 11.8X, and that changing multiple parameters can lead up to a 28X difference in performance. We find that these configuration parameters are often not set or reported in academic research, leading to incomplete and misleading evaluations. Running different off-the-shelf SQLite benchmarking tools such as Mobibench and Androbench in their default configuration shows upto 50% difference in performance. We hope this paper incites discussion in the systems community and among SQLite developers. We hope that our detailed analysis will help application developers to choose optimal SQLite parameters to achieve better performance.

1 Introduction

Benchmarks occupy an important place in the systems community: system design and development is driven by the results of carefully chosen and widely-agreed upon benchmarks. For example, the Yahoo Cloud Serving Benchmark [1] is used extensively to benchmark the performance of key-value stores such as LevelDB [2] and RocksDB [3], and the TPC benchmarks [4,5] are used to test the performance of databases and transaction processing systems. Architects base design decisions on such benchmarks. Industry makes purchasing decisions (worth millions of dollars) based on such benchmarks. Academic research also bases its results on such benchmarks. Thus, proper benchmarking is vital to the systems community.

Unfortunately, benchmarking complex systems in a repeatable fashion is a hard problem. Since benchmark results often depend upon a large number of factors both in the benchmark itself and in the operating system environment, producing repeatable results requires a great deal of care. Previous work by Mytkowicz *et al.* [6] and Tarasov *et al.* [7] has shown how a seemingly innocuous parameter in an experimental setup can lead to a significant bias in evaluating systems. They review some of the commonly used benchmarks to show how their performance differs by orders of magnitude [7].

Despite discussion in the systems community [6, 7], we believe two problems related to benchmarking are still rampant in industry and academia. First, there are several industrial tools which benchmark the same system and yet provide widely varying results. Second, when researchers report the results from benchmarks, they often omit important details about the setup without which it is impossible to reproduce their results. The combination of these problems lead to misleading results that cannot be directly compared, resulting in confusion.

To illustrate these points, we examine the benchmarking of SQLite [8], a lightweight, embedded relational database popular in mobile systems. SQLite is a commonly used benchmark in many mobile applications (such as Twitter and Facebook) to store their data [9, 10]. The SQLite website reports billions of deployments across different kinds of devices [11]. SQLite is also widely used in academic research. For example, SQLite has been used as a benchmark for evaluating new I/O scheduling frameworks [12], the Linux read-ahead mechanism [13], non-volatile write-ahead logging [14], and hardware-assisted fault tolerance [15]. Investigating the publications in the area of storage and systems in the past eight years, including papers from several premier conferences, we find that work involving SQLite has been done in 46 of these publications, with 25 papers published in the past two years [16]. Thus, benchmarking SQLite is an important part of evaluating these systems. In this paper, we focus on the evaluation of SQLite

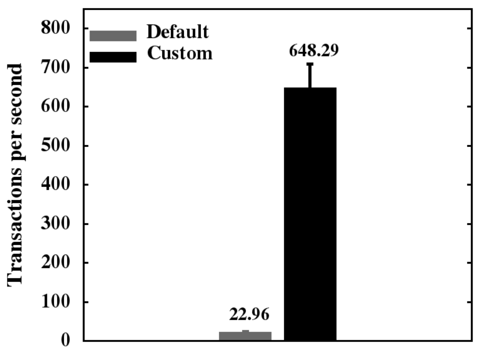


Figure 1: **Performance Impact of Parameters**. *The figure shows the 28X difference in performance from configuring SQLite parameters. Default config: journal mode `DELETE`, synch mode `FULL`, file system ext4. Custom config: journal mode `WAL`, `WAL` size 1 MB, synch mode `NORMAL`, file system F2FS.*

on the Android platform.

Evaluating SQLite performance is complex and error-prone. SQLite performance depends on a number of parameters, and as we show later, changing just *one* parameter results in performance differing by a staggering 11.8X (§3). Figure 1 shows how performance can differ by **28X**, by just varying a few parameters and switching the underlying file system from ext4 to F2FS.

We see that developers and researchers still do not report the required configuration parameters that affect the performance of SQLite. The popular tools available to benchmark SQLite give inconsistent results, often differing by 50% or more (§2) in its default configuration. When we investigated 16 papers from the past two years whose evaluation included SQLite, we find that **none** report all the parameters required to meaningfully compare results: ten papers do not report any parameters [17–26], five do not report the sync mode [27–31], while only one paper reports all parameters except write-ahead log size [32]. Without reporting how SQLite was configured, it is meaningless to compare SQLite results.

Most applications on Android use SQLite as the backend data-store with its default configuration, which is not performance efficient. It is important for the application developers to understand how different configuration parameters affect the performance of SQLite. We believe such understanding will be useful to academic researchers as well. Millions of Android users already use tools such as Mobibench to evaluate phones and storage devices. Thus, we believe that it is essential for researchers and developers to understand how to properly evaluate SQLite, and how to report results in a complete manner.

In this paper, we analyse the influence of different factors on SQLite performance (§3). The aim of our analysis is two fold. First, we intend our analysis to help application developers decide on an optimal SQLite configuration to achieve better performance. Second, we hope this paper serves as a wakeup call in the system community and improves the way in which benchmarking results (especially for SQLite) are presented in future.

Tool	**Default TPS**	**Custom TPS**
MobiBench	20	57
RLBench	30	-
Androbench	29	150

Table 1: **Benchmarking Tools**. *The table shows the throughput achieved by three SQLite benchmarking tools for 1000 insert operations. Note how the results differ by 50% in their default configuration, and as much as 3X after changing a single parameter.*

2 Tools to Benchmark SQLite

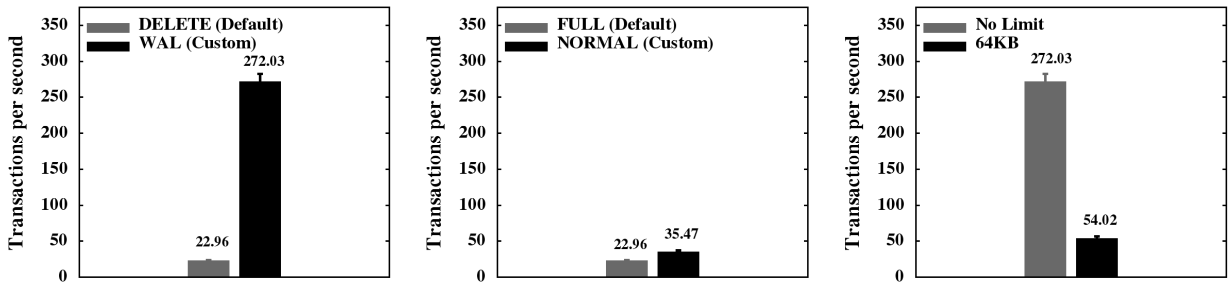
We examined 46 recent papers (published since 2008) that used SQLite. Several use benchmarking tools like Mobibench [33], RLbench [34], Androbench [35] to evaluate their system and application. A closer look at these publications reveal that seven of these works have used Mobibench to benchmark SQLite in their applications, making it one of the most sought after tools in the recent times for SQLite benchmarking. Four papers used RLbench and three papers used Androbench.

Table 1 shows how the performance of these benchmarks differ. In their default configurations, *on the same device*, RLbench and Androbench report 50% better throughput than Mobibench. Once a single parameter, the journaling mode, is switched from `DELETE` to `WAL`, Androbench achieves 3X the throughput of Mobibench.

Thus, it is easy to be mislead by the results of these benchmarking tools if the parameters are not set correctly. It is not enough if a tool becomes a widely used benchmark – for results to be compared, the relevant parameters should be reported.

3 Parameters Affecting Performance

We now describe the various parameters that can affect SQLite performance and how the performance varies with different settings. In order to evaluate this, we have developed an application in Android that allows us to configure different parameters for SQLite, as none of the existing benchmarking tools allow us to vary all of these parameters.



(a) Varying journal mode only (b) Varying synchronization mode only (c) Varying journal size only

Figure 2: **Performance impact of a single SQLite parameter change**. *The figure shows the 11.8X difference in performance due to changing only the journal mode, 1.5X difference due to varying the synchronization mode alone and a 5X change by modifying only the journal size.*

Hardware Setup. We run our experiments on a Samsung Galaxy Nexus S phone with Dual-core, 1.2GHz, Cortex-A9 processor; 32GB internal memory and 1GB RAM running Android 6.0.1 (cyanogenmod 13.0) on Linux 3.0.101 (F2FS enabled) kernel. The results presented in this paper are from experiments run on Android 6.0.1. Experiments were also performed on Android 4.2.1 to compare between different ROM versions, which is not presented.

We design experiments where we vary one parameter while keeping all other parameters constant. The default setting for SQLite is `DELETE` journal mode with `FULL` synchronization on Ext4 file system in ordered mode. We report the throughput (txs/sec) for performing 3000 transactions (1000 inserts, 1000 updates, and 1000 deletes). The experiments are performed on a database pre-populated with 100K rows. We report the average of ten runs.

We now describe the parameters that affect SQLite performance. Figure 2 shows the performance implication of changing some of these parameters, one parameter at a time.

SQLite Journaling mode. The SQLite journaling mode defines the type of journal used: `DELETE` (default), `TRUNCATE`, Write-Ahead Log (`WAL`), `PERSIST`, `MEMORY` and `OFF`. Figure 2a shows the drastic 11.8X improvement in performance by just changing the SQLite journal mode from `DELETE` to `WAL`. We later discuss how the SQLite journaling mode interacts with the file system journaling mode.

Synchronization Mode. SQLite synchronization modes controls the frequency at which `fsync()` command is issued by the SQLite library. The parameter has three values: `OFF`, `NORMAL`, and `FULL`. Figure 2b shows how performance increases by 54% when changing the synchronization mode from `FULL` to `NORMAL`.

SQLite Journal Size. This parameter is the size of the journal in SQLite, in bytes. Figure 2c illustrates how performance reduces by 5X when the journal size is reduced to 64 KB from the unlimited default setting.

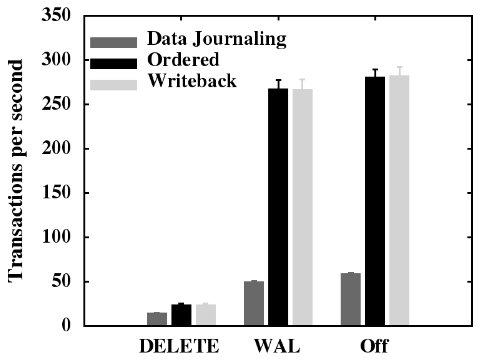
File System. The file system on the Android device can also impact the performance of applications that use SQLite, as it potentially changes the IO pattern at the block level. We present an analysis of how SQLite parameters interact with the file system type and the file system journaling mode.

Pre-populating the Database. To obtain realistic performance estimates, it is necessary to pre-populate the database on which the SQLite operations are performed. We discuss how not doing so can seem to increase the performance and why this has to be avoided.

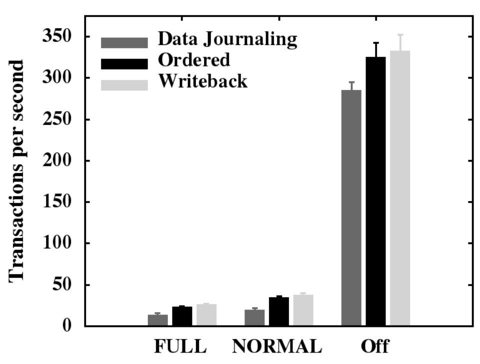
Apart from the parameters presented above, other factors such as the Flash Card used [36], the SQLite library version, and the Android version also affect performance. However, since the performance difference is small or well studied (as in the case of the flash card), we do not discuss such factors, but it is important to report these parameters while presenting benchmarking results.

3.1 SQLite Journal mode

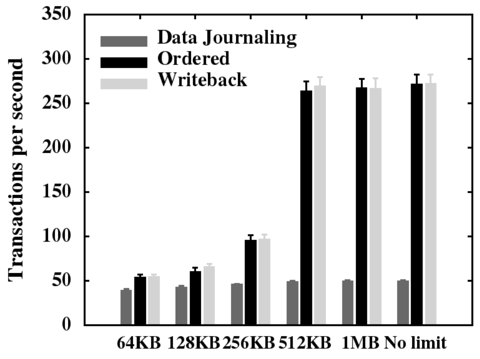
The different journal modes of SQLite are `DELETE`, `WAL`, `TRUNCATE`, `PERSIST`, `MEMORY` and `OFF`. The default journal mode is `DELETE`, which uses the traditional roll-back journaling mechanism in which the content of the database is written on to the journal and the changes are written to the database file directly. The journal is automatically committed and deleted at the end of each transaction. The `WAL` journal mode uses write-ahead log, in which the changes to the database are written to the journal and is committed to the database when the user externally triggers a commit or once a 1000 page limit is reached [37]. `TRUNCATE` mode is same as the `DELETE` mode, except that the roll-back journal is trun-



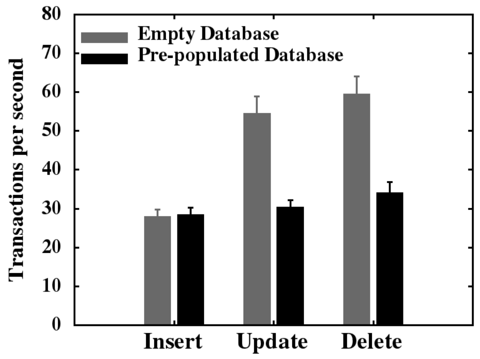
(a) **Performance for different SQLite journal modes.** *The figure shows that* `WAL` *mode performance is 10X higher than* `DELETE` *mode. Data journaling, ordered, and writeback are ext4 journaling modes.*



(b) **Performance for different synchronization modes.** *The figure shows that turning* `OFF` *SQLite synchronization gives the best performance, and* `NORMAL` *synch mode performs 50% better than* `FULL` *synch mode.*



(c) **Performance for different `WAL` sizes.** *The figure shows that the performance of SQLite increases as the* `WAL` *journal grows. Performance differs by 5X difference between* `WAL` *sizes of 64KB and 1 MB.*



(d) **Performance with pre-population.***The figure shows that the throughput decreases by 50% for update and delete operations when the database is pre-populated.*

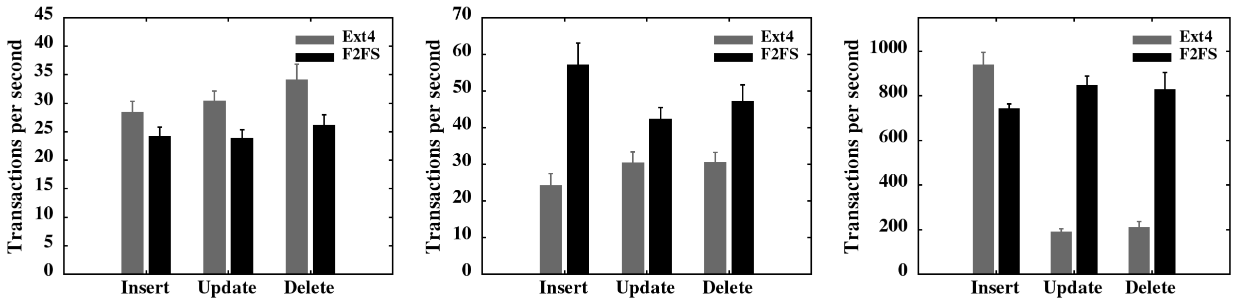
Figure 3: Impact of various parameters on SQLite performance

cated to zero length instead of deleting it after every transaction. In `MEMORY` mode, the roll-back journal is stored in-memory which reduces the disk IO at the cost of database safety and integrity. In a `PERSIST` mode, the roll-back journal is re-written after every transaction commit, which is same as truncating the journal on a flash device. SQLite also provides an option to turn off journaling, which improves performance by foregoing crash consistency.

In our experiments, we compare three most commonly used SQLite journaling modes – `DELETE`, `WAL` and `OFF` for different file system (ext4) journaling modes. Figure 3a presents the results of our experiments. We make two observations based on the results.

First, turning off journaling in SQLite increases performance, but at the cost of disabling the atomic commit and roll-back features of SQLite [38]. Write ahead log (`WAL`) mode outperforms `DELETE` mode by $\sim$10X in all file system configurations. This is because in `DELETE` mode, the roll-back journal is deleted at every transaction commit and hence an `fsync()` is initiated after every transaction. For a workload of 1000 SQLite inserts, 1000 `fsync()` calls were issued in `WAL` mode, while 5000 `fsync()` calls were issued in `DELETE` mode.

Second, as expected, the writeback file-system mode performs the best, while data journaling mode performs the worst [39]. Since data journaling generates the most IO (2X the IO generated by ordered mode), and writeback has the least ordering constraints, these results are consistent with expectations. We see the same pattern in other experiments as well.



(a) Journal:`DELETE`, Synch:`FULL` (b) Journal:`TRUNCATE`, Synch:`NORMAL` (c) Journal:`WAL`, Synch:`NORMAL`

Figure 4: **Performance on different filesystems**. *The figure shows how SQLite performance varies drastically (upto 20X) depending upon the underlying file system and the SQLite journal mode. Based on the journal mode, either ext4 or F2FS perform best for SQLite operations.*

3.2 SQLite Synchronization Mode

SQLite allows three modes of synchronization to persist the data written– `FULL`, `NORMAL` and `OFF`. `FULL` synchronization writes changes to the database on each commit, while `NORMAL` mode only writes to the log.In the `OFF` mode, dirty data is not forced to storage, and instead the operating system writes out the dirty data later. `OFF` mode can lead to corruption on crash, and hence is not preferred. Thus the three modes differ in the amount of IO and the number of `fsync()` calls issued. In practice, SQLite is run in `WAL` mode with `NORMAL` synchronization, to strike a balance between durability and performance.

Figure 3b shows the impact of modifying the synchronization mode in the default `DELETE` mode across Ext4 file system configurations. As expected, turning off all consistency mechanisms increases performance significantly. Note that while changing from `FULL` to `NORMAL` synch mode increases performance by 1.5X in `DELETE` mode, it increases performance by 3X in `WAL` mode.

3.3 SQLite Journal Size

In the `WAL` mode of SQLite, by default, there is no limit set on the journal size, which allows it to grow unbounded, thereby potentially affecting read performance [37]. Figure 3c shows the throughput across 1000 SQLite inserts, updates and deletes for different `WAL` sizes and different file system modes in ext4.

We observe that larger `WAL` sizes lead to higher throughput. There is a 5X performance difference between `WAL` sizes of 64 KB and 1 MB. When the `WAL` gets full, a checkpoint is triggered with an associated `fsync()` call. If the `WAL` is not full, a checkpoint is triggered every 1000 pages that are written to the `WAL`. For smaller `WAL` sizes, the `WAL` becomes full quickly, leading to performance degradation. Increasing the `WAL` size beyond 1000 pages does not help since checkpointing will be triggered for every 1000 pages anyway.

There is no performance benefit to increasing the `WAL` size beyond 1 MB, and bounding the `WAL` at this size helps maintain a trade-off between read and write performance of SQLite.

3.4 Pre-populating the Database

For many SQLite operations, the state of the database on which operations are performed, significantly affects it's performance. In the default configuration of SQLite (`DELETE` mode), when data is updated in an operation, the old data is first copied into a journal, and the new data is written directly into the database. Thus, whether the database already contains data or not significantly affects performance.

We conducted experiments where SQLite operations such as insert, update, and delete were performed both on an empty database, and a database pre-populated with 100K entries. Figure 3d shows the results.

We observe that for update and delete operations, performance is close to 2X higher if run on an empty database. Most benchmarking tools do not pre-populate the database, thus producing unrealistic performance numbers.

3.5 File Systems

The file system on which SQLite is run also impacts its performance, as application writes are transformed into different block IO by the file system. Having discussed how SQLite performance varies with ext4 journaling mode in previous sections, we now present how SQLite performance changes when run on a file system designed specifically for flash: F2FS [30].

Figure 4 shows the performance of different SQLite operations on ext4 and F2FS. We observe that in the default SQLite configuration, ext4 performs better than F2FS for all SQLite operations. In contrast, in the `TRUNCATE`/Synch-`NORMAL` configuration, F2FS outperforms ext4 in all SQLite operations. In the `WAL`/Synch-`NORMAL` configuration, SQLite updates and deletes perform $\sim$4X better in F2FS, whereas SQLite inserts are 30% faster in ext4. Thus, depending on the experiment you choose, you can show either F2FS or ext4 performing better for SQLite operations! We note that the F2FS paper only reports performance on `WAL` mode, without reporting the synchronization setting [30].

We also observed a 6X improvement in performance when the synchronization mode is changed from `FULL` To `NORMAL` in `WAL` journal mode in F2FS.

4 Conclusion

We inspect the state of benchmarking (and how results are reported) in industry and academia, using SQLite as an example. We show that benchmarking SQLite is complex and that SQLite performance depends upon a large number of parameters. Tuning a few parameters can vary performance significantly (upto 28X). We show that industrial tools for benchmarking SQLite report widely varying results, and that academic research does not report on all the configuration parameters required to reproduce benchmarking results.

We hope our study accomplishes two objectives. First, we hope it helps developers and researchers realize the impact of these parameters on SQLite performance. Second, and more importantly, we hope it spurs discussion about what must be reported in conjunction with SQLite results to make the results repeatable and comparable. Given the importance of benchmarks in the systems community and the rising use of SQLite, we believe this discussion is crucial and timely.

5 Acknowledgments

We would like to thank the anonymous reviewers for their feedback and guidance. This work was supported by generous donations from VMware and Facebook. Any opinions, findings, and conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of other institutions.

References

[1] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[2] Google. Leveldb. `https://github.com/google/leveldb`.

[3] RocksDB — A persistent key-value store. `http://rocksdb.org`.

[4] Transaction Processing Performance Council. Tpc benchmark c, standard specification version 5, 2001.

[5] Transaction Processing Performance Council. ‘tpc benchmark b. *Standard Specification, Waterside Associates, Fremont, CA*, 1990.

[6] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 265–276, New York, NY, USA, 2009. ACM.

[7] Vasily Tarasov, Saumitra Bhanage, Erez Zadok, and Margo Seltzer. Benchmarking file system benchmarking: It* is* rocket science. In *HotOS*, 2011.

[8] SQLite. SQLite transactional SQL database engine. `http://www.sqlite.org/`.

[9] Why to use SQLite in Android. `https://www.quora.com/Why-should-we-use-SQLite-in-Android-development`, January 2017.

[10] SQLite in Android. `http://www.grokkingandroid.com/sqlite-in-android`, January 2017.

[11] SQLite. Most Widely Deployed SQL Database Engine. `https://www.sqlite.org/mostdeployed.html`.

[12] Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini T Kaushik, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Split-level i/o scheduling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 474–489. ACM, 2015.

[13] Pierre Olivier, Jalil Boukhobza, and Eric Senn. Revisiting read-ahead efficiency for raw nand flash storage in embedded linux. *ACM SIGBED Review*, 11(4):43–48, 2015.

[14] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. Nvwal: Exploiting nvram in write-ahead logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 385–398. ACM, 2016.

[15] Dmitrii Kuvaiskii, Rasha Faqeh, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Haft: hardware-assisted fault tolerance. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 25. ACM, 2016.

[16] Publications involving SQLite. `http://dl.acm.org/results.cfm?query=(sqlite)&within=owners.owner=HOSTED&filtered=&dte=2008&bfr=`, January 2017.

[17] Nikolaos Ch. Kasmeridis and Michael Gr. Vassilakopoulos. A diet-guide mobile application for diabetes mellitus management. In *Proceedings of the 19th Panhellenic Conference on Informatics*, PCI ’15, pages 377–381, New York, NY, USA, 2015. ACM.

[18] Simone Mutti, Enrico Bacis, and Stefano Paraboschi. Sesqlite: Security enhanced sqlite: Mandatory access control for android databases. In *Proceedings of the 31st Annual Computer Security Applications Conference*, ACSAC 2015, pages 411–420, New York, NY, USA, 2015. ACM.

[19] Eunryoung Lim, Seongjin Lee, and Youjip Won. Androtrace: Framework for tracing and analyzing ios on android. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW ’15, pages 3:1–3:8, New York, NY, USA, 2015. ACM.

[20] Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini T. Kaushik, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Split-level i/o scheduling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP ’15, pages 474–489, New York, NY, USA, 2015. ACM.

[21] Dmitrii Kuvaiskii, Rasha Faqeh, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Haft: Hardware-assisted fault tolerance. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 25:1–25:17, New York, NY, USA, 2016. ACM.

[22] Charlie Curtsinger and Emery D. Berger. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 184–197, New York, NY, USA, 2015. ACM.

[23] Yan Wang and Atanas Rountev. Profiling the responsiveness of android applications via automated resource amplification. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, MOBILESoft '16, pages 48–58, New York, NY, USA, 2016. ACM.

[24] Georgios Chatzopoulos, Aleksandar Dragojević, and Rachid Guerraoui. Estima: Extrapolating scalability of in-memory applications. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 27:1–27:11, New York, NY, USA, 2016. ACM.

[25] Pierre Olivier, Jalil Boukhobza, and Eric Senn. Revisiting read-ahead efficiency for raw nand flash storage in embedded linux. *SIGBED Rev.*, 11(4):43–48, January 2015.

[26] Phil Mcminn, Chris J. Wright, and Gregory M. Kapfhammer. The effectiveness of test coverage criteria for relational database schema integrity constraints. *ACM Trans. Softw. Eng. Methodol.*, 25(1):8:1–8:49, December 2015.

[27] Lei Li, Kai Qian, Qian Chen, Ragib Hasan, and Guifeng Shao. Developing hands-on labware for emerging database security. In *Proceedings of the 17th Annual Conference on Information Technology Education*, SIGITE '16, pages 60–64, New York, NY, USA, 2016. ACM.

[28] Gihwan Oh, Sangchul Kim, Sang-Won Lee, and Bongki Moon. Sqlite optimization with phase change memory for mobile applications. *Proc. VLDB Endow.*, 8(12):1454–1465, August 2015.

[29] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. Nvwal: Exploiting nvram in write-ahead logging. *SIGOPS Oper. Syst. Rev.*, 50(2):385–398, March 2016.

[30] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. F2fs: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 273–286, Berkeley, CA, USA, 2015. USENIX Association.

[31] Eric Koskinen and Junfeng Yang. Reducing crash recoverability to reachability. *SIGPLAN Not.*, 51(1):97–108, January 2016.

[32] Dam Quang Tuan, Seungyong Cheon, and Youjip Won. On the io characteristics of the sqlite transactions. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, MOBILESoft '16, pages 214–224, New York, NY, USA, 2016. ACM.

[33] Sooman Jeong, Kisung Lee, Jungwoo Hwang, Seongjin Lee, and Youjip Won. Androstep: Android storage performance analysis tool. In *Software Engineering (Workshops)*, volume 13, pages 327–340, 2013.

[34] RL Benchmark. `https://play.google.com/store/apps/details?id=com.redlicense.benchmark.sqlite&hl=en`, December 2016.

[35] Je-Min Kim and Jin-Soo Kim. Androbench: Benchmarking the storage performance of android-based mobile devices. In *Frontiers in Computer Education*, pages 667–674. Springer, 2012.

[36] Hyojun Kim, Nitin Agrawal, and Cristian Ungureanu. Revisiting storage for smartphones. *Trans. Storage*, 8(4):14:1–14:25, December 2012.

[37] WAL mode in SQLite. `https://www.sqlite.org/wal.html`, January 2017.

[38] No journal in SQLite. `https://www.sqlite.org/pragma.html#pragma_journal_mode`, January 2017.

[39] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *The Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, pages 105–120, Anaheim, CA, April 2005.